What's in a name?

Représenter les variables et leurs liaisons

Xavier Leroy

2020-02-13

Collège de France, chaire de sciences du logiciel

What's in a name? that which we call a rose
By any other name would smell as sweet.

W. Shakespeare, Romeo and Juliet

2.1.12. CONVENTION. *Terms that are α-congruence are identified. So now we write $\lambda x.x \equiv \lambda y.y$, etcetera.*

H. Barendregt, The Lambda Calculus, 1984.

We thank T. Thacher Robinson for showing us on August 19, 1962 by a counterexample the existence of an error in our handling of bound variables.

S. C. Kleene, Disjunction and Existence Under Implication in Elementary Intuitionistic Formalisms, 1962.

Variables liées, renommage, et alpha-conversion

La plupart des langages de programmation ont des constructions qui lient des variables :

- Définitions de fonctions : $\lambda x.\ M$
- Définitions de variables locales : $\texttt{let}\ x = M\ \texttt{in}\ N$
- Filtrages : $\texttt{match}\ M\ \texttt{with}\ (x, y) \to N$
- Quantificateurs (dans les types) : $\forall \alpha.\ \alpha \to \alpha$

On peut renommer les variables liées sans changer le sens :

$$\lambda x.\ x + 1 \equiv \lambda y.\ y + 1 \qquad\qquad \forall \alpha,\ \alpha \to \alpha \equiv \forall \beta,\ \beta \to \beta$$

Ce renommage s'appelle l'alpha-conversion.

Substitution, lieurs, et capture

En présence de lieurs (constructions qui lient des variables), la substitution $M\{x \leftarrow N\}$ d'une variable x par un terme N doit éviter de capturer une variable libre dans N par un lieur de M :

$$\begin{aligned} (\lambda y.\, x + y)\{x \leftarrow z\} &= \lambda y.\, z + y \quad \checkmark \\ (\lambda y.\, x + y)\{x \leftarrow y\} &= \lambda y.\, y + y \quad \times \end{aligned}$$

Dans le second cas, la variable libre y est capturée par le lieur λy.

On évite la capture par alpha-conversion avant substitution :

$$(\lambda y.\, x + y)\{x \leftarrow y\} = (\lambda u.\, x + u)\{x \leftarrow y\} = \lambda u.\, y + u \quad \checkmark$$

Formaliser un langage avec des lieurs

Sur le papier, l'usage est de prendre les termes du langage modulo alpha-conversion.

1. On définit une syntaxe abstraite avec des noms et des lieurs.

$$M, N ::= x \mid \lambda x.\, M \mid M\ N$$

2. On définit une relation $\equiv$ d'égalité modulo alpha-conversion

$$M \equiv M \qquad \frac{y \notin \mathcal{L}(M) \quad M\{x \leftarrow y\} \equiv N}{\lambda x.\, M \equiv \lambda y.\, N} \qquad \frac{M \equiv M' \quad N \equiv N'}{M\ N \equiv M'\ N'}$$

3. On travaille dans l'ensemble quotient $\mathit{Termes}/\equiv$

Formaliser un langage avec des lieurs

Sur le papier, l'usage est de prendre les termes du langage modulo alpha-conversion.

Cet usage est difficile à mécaniser avec un assistant à la démonstration.

Pour des langages simples (réductions faibles, types simples) on peut travailler sans alpha-conversion.
(Ex : le cours du 6 février et le développement Coq `FUN.v`).

Sinon, il faut inventer d'autres manières de décrire les variables et leurs lieurs. (Ce séminaire).

Les difficultés de l'alpha-conversion

1. Travailler dans un ensemble quotient.

$$M \equiv M \qquad \frac{y \notin \mathcal{L}(M) \quad M\{x \leftarrow y\} \equiv N}{\lambda x.\, M \equiv \lambda y.\, N} \qquad \frac{M \equiv M' \quad N \equiv N'}{M\ N \equiv M'\ N'}$$

Toutes les définitions et tous les énoncés doivent être compatibles avec cette relation d'équivalence $\equiv$.

Exemple (La définition des variables libres dans un terme)

$$\mathcal{L}(x) = \{x\} \quad \mathcal{L}(\lambda x.\, M) = \mathcal{L}(M) \setminus \{x\} \quad \mathcal{L}(M\ N) = \mathcal{L}(M) \cup \mathcal{L}(N)$$

Il faut démontrer que $\mathcal{L}(M) = \mathcal{L}(N)$ si $M \equiv N$.

Les difficultés de l'alpha-conversion

2. Définir les bons principes de récurrence.

On démontre une propriété $P(M)$ d'un terme M par récurrence sur la structure du terme M.

Dans le cas $P(\lambda x.\ M)$, quelle hypothèse de récurrence peut-on supposer ?

- uniquement que $P(M)$ est vraie ?
- ou bien que $P(M\{x \leftarrow y\})$ est vraie pour tout $y \notin \mathcal{L}(M)$?

Les difficultés de l'alpha-conversion

3. Définir la substitution sans captures.

$$(\lambda y.\, M)\{x \leftarrow N\} = \lambda z.\, (M\{y \leftarrow z\})\, \{x \leftarrow N\} \quad \text{«si } z \text{ est fraîche»}$$

Comment choisir z de manière appropriée et déterministe ?
P.ex. z est le plus petit nom de variable non libre dans M ni N.

Comment traiter cette récurrence qui n'est pas structurelle
($M\{y \leftarrow z\}$ n'est pas un sous-terme de $\lambda y.\, M$)
et donc pas acceptée directement par Coq ?

Il faut utiliser d'autres styles de définition : récurrence sur la taille du terme M, prédicat inductif au lieu de fonction, …

Mécaniser des langages avec variables et lieurs

Pour contourner ces difficultés avec l'alpha-conversion, plusieurs approches ont été proposées et utilisées. Nous allons en décrire quatre :

1. La notation de de Bruijn.
2. La syntaxe abstraite d'ordre supérieur (HOAS, *Higher Order Abstract Syntax*).
3. Les logiques nominales.
4. L'approche ≪sans noms locaux≫ (*locally nameless*).

Le *POPLmark challenge* (U. Penn., 2005) compare ces approches et d'autres sur un même problème : démontrer la sûreté du système de types $F_{<:}$ (≪F-sub≫).

La notation de de Bruijn

La notation de de Bruijn

(N. de Bruijn, le système Automath, 1967 ; *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation*, 1972.)

$$
\begin{array}{rll}
M, N ::= & \underline{1} \mid \underline{2} \mid \cdots \mid \underline{n} \mid \cdots & \text{variable} \\
& \mid \lambda.M & \text{abstraction sur la variable } \underline{1} \\
& \mid M\ N & \text{application}
\end{array}
$$

Identifie chaque variable pas par un nom, mais par sa position par-rapport à la lambda-abstraction qui la lie.

$$\lambda x.\ \lambda y.\ x\ (y\ y) \quad \equiv \quad \lambda.\lambda.\ \underline{2}\ (\underline{1}\ \underline{1}) \quad \equiv \quad \lambda u.\ \lambda v.\ u\ (v\ v)$$

Une représentation canonique : deux termes sont alpha-convertibles si et seulement si leurs représentation en de Bruijn sont égales.

La substitution en notation de de Bruijn

$$
\begin{aligned}
\underline{x}\,\{n \leftarrow P\} &= \begin{cases} \underline{x} & \text{si } x < n \\ P & \text{si } x = n \\ \underline{x-1} & \text{si } x > n \end{cases} \\
(\lambda.M)\,\{n \leftarrow P\} &= \lambda.\, M\{n+1 \leftarrow \Uparrow_1 P\} \\
(M\,N)\,\{n \leftarrow P\} &= M\{n \leftarrow P\}\, N\{n \leftarrow P\}
\end{aligned}
$$

Cas des variables : substituer $\underline{n}$ correspond à supprimer le n-ième λ englobant. Donc, $\underline{n+1}$ devient $\underline{n}$, $\underline{n+2}$ devient $\underline{n+1}$, etc.

Cas des abstractions : pour éviter la capture, il faut recaler (incrémenter) tous les indices des variables libres de P.

Le recalage (*lifting*) des indices de de Bruijn

$$
\begin{aligned}
\Uparrow_n \underline{x} &= \begin{cases} \underline{x} & \text{si } x < n \\ \underline{x+1} & \text{si } x \geq n \end{cases} \\
\Uparrow_n (\lambda.M) &= \lambda.\ \Uparrow_{n+1} M \\
\Uparrow_n (M\ N) &= \Uparrow_n M\ \Uparrow_n N
\end{aligned}
$$

$\Uparrow_n M$ recale (incrémente les indices) des variables libres dans M.
n est l'indice de la première variable libre.

Propriétés de la substitution et du décalage

Quelques lemmes techniques de commutation : si $p \geq n$,

$$\begin{aligned}
(\Uparrow_n M)\{n \leftarrow N\} &= M \\
\Uparrow_n (M\{p \leftarrow N\}) &= (\Uparrow_n M)\{p+1 \leftarrow \Uparrow_n N\} \\
\Uparrow_p (M\{n \leftarrow N\}) &= (\Uparrow_{p+1} M)\{n \leftarrow \Uparrow_p N\} \\
M\{n \leftarrow N\}\{p \leftarrow L\} &= M\{p+1 \leftarrow \Uparrow_n L\}\{n \leftarrow N\{p \leftarrow L\}\}
\end{aligned}$$

En notation nommée usuelle, le dernier lemme s'écrit
$M\{x \leftarrow N\}\{y \leftarrow L\} = M\{y \leftarrow L\}\{x \leftarrow N\{y \leftarrow L\}\}$
à condition que $x \neq y$ et x pas libre dans L.

Avantage de de Bruijn : systématique, pas de conditions.
Inconvénient : peu lisible, peu intuitif.

Énoncer des théorèmes en notation de de Bruijn

Les énoncés de propriétés usuelles incluent des recalages d'indices pour les variables libres. Cela les rend plus compliqués que les énoncés usuels.

Exemple (Le lemme d'affaiblissement)

En notation usuelle : $E, E' \vdash M : \tau \implies E, x : \sigma, E' \vdash M : \tau$

Même M en conclusion. Hypothèse que x est non lié dans E, E'.

En notation de de Bruijn : $E', E \vdash M : \tau \implies E', \sigma, E \vdash \Uparrow_{|E'|+1} M : \tau$

Il faut recaler dans M les variables liées «avant» x, c'est-à-dire les variables $> |E'|$. Pas d'hypothèses supplémentaires.

Énoncer des théorèmes en notation de de Bruijn

Exemple (La stabilité du typage par substitution)

L'énoncé du théorème final est clair :

$$\sigma, E \vdash M : \tau \quad \wedge \quad E \vdash N : \sigma \quad \Longrightarrow \quad E \vdash M\{1 \leftarrow N\} : \tau$$

Pour le démontrer par récurrence sur M, il faut renforcer l'énoncé et c'est moins clair :

$$E', \sigma, E \vdash M : \tau \quad \wedge \quad E', E \vdash N : \sigma \quad \Longrightarrow \quad E \vdash M\{|E'| + 1 \leftarrow N\} : \tau$$

Point d'étape

La notation de de Bruijn est une solution rigoureuse et fiable au problème de mécaniser des langages avec lieurs.

Les définitions et les énoncés des théorèmes sont différents de ce qu'on écrit ≪sur papier≫ avec des variables nommées. Il faut du temps pour s'habituer.

Beaucoup d'≪échafaudage≫ (définitions et propriétés des recalages, des substitutions, des environnements), mais qui peut être autogénéré (cf. le système Autosubst de S. Schäfer et al).

La syntaxe abstraite d'ordre supérieur

La syntaxe abstraite d'ordre supérieur (HOAS)

(D. Miller et G. Nadathur, *A Logic Programming Approach to Manipulating Formulas and Programs*, 1987.

F. Pfenning et C. Elliott, *Higher-order abstract syntax*, 1988.)

Idée : représenter les lieurs du langage ≪objet≫ par des fonctions du langage ≪hôte≫.

S.A. du 1er ordre :

```
type term =
  | Var of name
  | Abs of name * term
  | App of term * term
```

--->

S.A. d'ordre supérieur :

```
type term =

  | Abs of term -> term
  | App of term * term
```

Programmer avec la HOAS

```
type term =
  | Abs of term -> term
  | App of term * term
```

Il n'y a plus de cas `Var`, car les variables sont représentées par des méta-variables du langage hôte. Par exemple, $\lambda x.\lambda y.\ x\ y$ est

```
Abs(fun x -> Abs(fun y -> App(x, y)))
```

La substitution (du paramètre formel par l'argument effectif dans le corps de la fonction) est juste l'application de fonction !

```
let beta_reduction = function
  | App (Abs f, arg) -> f arg
```

Un lambda pour les lier tous

Dans les années 1930, Church avait introduit la lambda-notation dans l'espoir d'unifier de nombreuses notations mathématiques qui lient des variables :

$$\begin{aligned}
\forall x, P(x) &\equiv \forall(\lambda x.\, P(x)) \quad \text{(ou juste } \forall P\text{)} \\
\exists x, P(x) &\equiv \exists(\lambda x.\, P(x)) \\
\sum_{n=0}^{\infty} \frac{1}{n^2} &\equiv \mathtt{sum}(0, \infty, \lambda n. \frac{1}{n^2}) \\
\int_a^b f(x)dx &\equiv \mathtt{integr}(a, b, \lambda x. f(x))
\end{aligned}$$

HOAS suit la même approche : au lieu d'un constructeur $\mathtt{Abs}(x, M)$ qui lie x dans M de manière ad-hoc, on utilise le lieur universel λ et on écrit $\mathtt{Abs}(\lambda x.M)$.

Raisonner avec la HOAS

Comment raisonner sur des termes non clos ? En utilisant l'implication et le contexte de la logique hôte !

Exemple (Règles de typage simple)

$$\frac{\texttt{assm}\ x\ \tau}{\vdash x : \tau} \qquad \frac{\forall x,\ \texttt{assm}\ x\ \sigma \Rightarrow\ \vdash f\ x : \tau}{\vdash \texttt{Abs}\ f : \sigma \to \tau}$$

`assm` est un axiome de type $\texttt{term} \to \texttt{type} \to \texttt{Prop}$.

Le contexte de typage est représenté par l'ensemble des hypothèses $\texttt{assm}\ x\ \tau$ disponibles dans le contexte de la logique hôte.

Raisonner avec la HOAS

Exemple de dérivation de typage :

$$\dfrac{\dfrac{\dfrac{}{\texttt{assm}\ x\ \tau}}{\forall y,\ \texttt{assm}\ y\ \sigma \Rightarrow\ \vdash x : \tau}}{\forall x,\ \texttt{assm}\ x\ \tau \Rightarrow \vdash \texttt{Abs}(\texttt{fun}\ y \to x) : \sigma \to \tau}}{\vdash \texttt{Abs}\ (\texttt{fun}\ x \to \texttt{Abs}(\texttt{fun}\ y \to x)) : \tau \to \sigma \to \tau}$$

Note : on obtient «gratuitement» la stabilité du typage par affaiblissement du contexte, permutation d'hypothèses, etc.

Note : on ne peut pas avoir de contexte contradictoire comme $\texttt{assm}\ x\ \tau, \texttt{assm}\ x\ \sigma$.

Problème : les termes ≪exotiques≫

En Caml comme en Coq, le type `term -> term` contient bien plus de fonctions que les seuls codages HOAS de lambda-termes.

Un codage HOAS est une fonction paramétrique en son argument. Par exemple, elle est soit constante soit injective :

$$(\forall MN,\ f\ M = f\ N)\ \vee\ (\forall MN, f\ M = f\ N \Rightarrow M = N)$$

Une fonction Caml ou Coq peut discriminer sur son argument :

```
f = fun x -> match x with App _ -> x | Lam _ -> App(x, x)
```

Une telle fonction n'est pas paramétrique :

```
f (Lam g) = App(Lam g, Lam g) = f (App (Lam g, Lam g))
```

Problème : des types non inductifs

```
Inductive term : Type :=
  | Lam (f: term -> term)
  | App (a: term) (b: term).
```

Cette déclaration est rejetée par Coq ou Agda comme non inductive, à cause de l'occurrence de `term` en position négative dans le type du constructeur `Lam`.

Un tel type non inductif permettrait de définir des calculs qui ne terminent pas :

```
Definition delta (t: term) : term :=
  match t with Lam f => f t | _ => t end.
Definition omega : term := delta (Lam delta).
```

Le système Twelf

(F. Pfenning, C. Schurmann et al, *Twelf*, `http://twelf.org/`)

Un *logical framework* pour définir et raisonner sur des langages et des logiques exprimées en style HOAS.

- Les termes du langage objet sont représentés dans le langage LF
  $\approx$ lambda-abstraction, application, constructeurs.
  (Mais pas de destructeurs $\Rightarrow$ pas de termes exotiques.)
- Les propriétés des termes (p.ex. le jugement de typage) et les méta-propriétés (p.ex. la sûreté du typage) sont exprimées par des prédicats inductifs.

(Extensions de l'approche : Delphin, Abella, Beluga)

Approches nominales

Les logiques nominales

(M. J. Gabbay, A. M. Pitts, *A new approach to abstract syntax with variable binding*, 2002.
A. M. Pitts, *Nominal logic, a first-order theory of names and bindings*, 2003.
M. J. Gabbay, *A study of substitution, using nominal techniques and Fraenkel-Mostowski sets*, 2009.)

Des logiques où

- les notions de nom, de renommage, et de liaisons de noms sont primitives ;
- toutes les définitions et toutes les propositions sont équivariantes, c.à.d. invariantes par renommage.

Les approches nominales

(Ch. Urban, *Nominal techniques in Isabelle/HOL*, 2008)

Exprimer dans une logique mécanisée existante les principales notions de la logique nominale (noms, renommages, équivariance), typiquement sous forme de classe de types.

Exemple (Le package Nominal pour Isabelle/HOL)

```
atom_decl name
nominal_datatype lam = Var "name"
                     | App "lam × lam"
                     | Lam "≪name≫ lam"
```

Définit automatiquement une représentation `lam` où les termes alpha-convertibles sont égaux, ainsi que les principes de récurrence adaptés.

Noms et échanges

Il est commode de définir l'alpha-conversion non pas en termes de renommages $\{x \leftarrow y\}$ mais en termes d'échanges (*swaps*) (permutation de deux noms).

$$\binom{x}{y} \stackrel{def}{=} \{x \leftarrow y; y \leftarrow x\}$$

Au contraire d'un renommage, un échange est une bijection (c'est son propre inverse) et peut s'appliquer uniformément aux variables libres et aux variables liées, sans risque de capture :

$$\binom{x}{y}(\lambda z.M) = \lambda \binom{x}{y} z.\ \binom{x}{y} M$$

Types nominaux

Un type nominal est un type T muni d'une opération d'échange

$$\binom{x}{y} : T \to T$$

qui satisfait les propriétés attendues pour un échange de noms :

$$\binom{x}{x} t = t \qquad \binom{x}{y}\binom{x}{y} t = t \qquad \binom{x}{y}\binom{u}{v} t = \binom{\binom{x}{y} u}{\binom{x}{y} v}\binom{x}{y} t$$

De nombreux types sont nominaux :

- le type des noms ; les types sans noms (entiers, booléens) ;
- les produits, sommes, options, listes de types nominaux ;
- les fonctions entre types nominaux.

Support et fraîcheur

(Une abstraction de la notion de variable libre ou non libre.)

Pour tout type nominal, on peut définir des notions abstraites de

- Occurence (un nom apparaît dans un terme)

$$x \in t \iff \binom{x}{y} t \neq t \text{ pour une infinité de } y$$

- Fraîcheur (un nom n'apparaît pas dans un terme)

$$x\#t \iff x \notin t \iff \binom{x}{y} t = t \text{ pour une infinité de } y$$

- Support (tous les noms apparaissant dans un terme)

$$supp(t) = \{x \mid x \in t\}$$

Types à supports finis

$$
\begin{aligned}
supp(x) &= \{x\} \\
supp((t_1, t_2)) &= supp(t_1) \cup supp(t_2) \\
supp([t_1; \ldots; t_n]) &= supp(t_1) \cup \cdots \cup supp(t_n)
\end{aligned}
$$

On s'intéresse particulièrement aux types T à support fini : pour tout $t \in T$, l'ensemble $supp(t)$ est fini, et on peut toujours choisir un nom ≪frais≫ x tel que $x\#t$.

Support fini : noms, produits, sommes, listes, …
Support pas toujours fini : fonctions $T_1 \to T_2$.

Lier un nom

$[x]t$ représente le terme $t : T$ dans lequel on lie le nom x.
(P.ex. $\lambda x.t$ ou $\forall x.t$ ou …)

Il est caractérisé par :

$$[x]t = [x']t' \quad \text{ssi} \quad (x = x' \wedge t = t') \vee (x \neq x' \wedge t = \binom{x}{x'} t' \wedge x\#t')$$

$$y\#[x]t \quad \text{ssi} \quad y = x \vee y\#t$$

On peut le définir comme une fonction des noms dans `option` T :

$$\begin{aligned}[x]t = \lambda y.\ & \texttt{if } x = y \texttt{ then Some}(t) \\ & \texttt{else if } y\#t \texttt{ then Some}(\tbinom{x}{y} t) \\ & \texttt{else None}\end{aligned}$$

Comment quantifier les noms frais ?

Exemple : la règle de typage d'une abstraction $\mathtt{Abs}([x]M)$.

$$\frac{\forall x, x\#(E, M) \Rightarrow E, x : \sigma \vdash M[x] : \tau}{E \vdash \mathtt{Abs}([x]M) : \sigma \to \tau} \text{ (universellement)}$$

$$\frac{\exists x, x\#(E, M) \wedge E, x : \sigma \vdash M[x] : \tau}{E \vdash \mathtt{Abs}([x]M) : \sigma \to \tau} \text{ (existentiellement)}$$

Quantification «pour tout x frais» : principe de récurrence puissant ; introduction difficile.

Quantification «il existe un x frais» : introduction facile, principe de récurrence faible.

Les deux quantifications sont équivalentes !

Le quantificateur de fraîcheur

Soit $\varphi(x, \vec{t})$ une formule de logique nominale qui dépend d'un nom x et de termes nominaux $\vec{t}$.

Le quantificateur **И**x («pour tout x frais») :

$$\textbf{И}x.\ \varphi(x, \vec{t}) \stackrel{def}{=} \forall x\#\vec{t}.\ \varphi(x, \vec{t}) \Leftrightarrow \exists x\#\vec{t}.\ \varphi(x, \vec{t})$$

Dans la logique nominale de Pitts, toutes les formules logiques φ sont invariantes par échange de nom. Donc, si $\varphi(x, \vec{t})$ est vraie pour un certain $x\#\vec{t}$, pour tout autre $x'\#\vec{t}$ on a

$$\begin{pmatrix} x \\ x' \end{pmatrix} \varphi(x, \vec{t}) \Leftrightarrow \varphi(\begin{pmatrix} x \\ x' \end{pmatrix} x, \begin{pmatrix} x \\ x' \end{pmatrix} \vec{t}) = \varphi(x', \vec{t})$$

En Nominal Isabelle/HOL, toutes les formules ne sont pas invariantes. L'équivalence $\forall$-$\exists$ fait partie du principe de récurrence engendré automatiquement.

Le quantificateur de fraîcheur

Application : la règle de typage d'une abstraction $\texttt{Abs}([x]M)$.

$$\frac{\text{И}x.\ E, x : \sigma \vdash M[x] : \tau}{E \vdash \texttt{Abs}([x]M) : \sigma \to \tau}$$

Pour conclure $E \vdash \texttt{Abs}([x]M) : \sigma \to \tau$ il suffit de trouver un seul x frais qui vérifie la prémisse.

À partir de l'hypothèse $E \vdash \texttt{Abs}([x]M) : \sigma \to \tau$ on peut supposer la prémisse vraie pour une infinité de x frais.

L’approche «sans noms locaux»

L'approche sans noms locaux (*locally nameless*)

(McKinna et Pollack, *Pure Type Systems formalized*, 1993.
Aydemir, Charguéraud, Pierce, Pollack, Weirich, *Engineering Formal Metatheory*, 2008)

Idée 1 : représenter différement les variables libres et les variables liées ($\Rightarrow$ moins de problèmes de capture).

Idée 2 : représenter les variables libres par des noms (plus lisible) et les variables liées par des indices de de Bruijn (canonique).

L'approche sans noms locaux

$$
\begin{array}{rll}
M, N ::= & \underline{n} & \text{variable liée, d'indice } n \\
| & x & \text{variable libre, de nom } x \\
| & \lambda.M & \text{abstraction sur la variable } \underline{1} \\
| & M\ N & \text{application}
\end{array}
$$

Exemple : $\lambda x.\ x\ y$ est $\lambda.\ \underline{1}\ y$.

Invariant : les termes sont toujours localement clos : pas d'indice de de Bruijn libre.

Substitutions

Deux opérations de substitution :

- d'un nom par un terme $\{x \leftarrow N\}$

$$(\lambda.M)\{x \leftarrow N\} = \lambda.\, M\{x \leftarrow N\}$$

- d'un indice par un terme $[n \leftarrow N]$

$$(\lambda.M)\{n \leftarrow N\} = \lambda.\, M\{n+1 \leftarrow N\}$$

Pas de risque de capture :

parce que N est localement clos ;

parce que λ ne lie aucun nom.

(Il faut encore prouver des lemmes de commutation, mais c'est plus simple que dans l'approche indices de de Bruijn.)

Traverser les lieurs

Comment ≪récurser≫ dans un terme $\lambda.M$?

- Non : récurser sur M (pas localement clos !)
- Oui : récurser sur $M^x \stackrel{def}{=} M[1 \leftarrow x]$ avec x un nom ≪frais≫.

Exemple (La règle de typage pour l'abstraction)

$$\frac{x \notin Dom(E) \cup FV(M) \qquad E, x : \sigma \vdash M^x : \tau}{E \vdash \lambda.M : \sigma \to \tau}$$

Comment quantifier les noms frais ?

$$\frac{\forall x, x \notin Dom(E) \cup FV(M) \Rightarrow E, x : \sigma \vdash M^x : \tau}{E \vdash \lambda.M : \sigma \to \tau} \text{ (universellement)}$$

$$\frac{\exists x, x \notin Dom(E) \cup FV(M) \wedge E, x : \sigma \vdash M^x : \tau}{E \vdash \lambda.M : \sigma \to \tau} \text{ (existentiellement)}$$

Même dilemme que dans l'approche nominale : $\exists x$ est plus commode pour l'introduction, mais $\forall x$ donne un principe de récurrence plus fort.

Les deux quantifications sont équivalentes à condition de montrer que le prédicat $E \vdash M : \tau$ est équivariant (invariant par échanges). C'est beaucoup de travail…

Quantification cofinie

(Aydemir, Charguéraud, Pierce, Pollack, Weirich, *Engineering Formal Metatheory*, 2008)

$$\frac{\exists L, \forall x, x \notin L \Rightarrow E, x : \sigma \vdash M^x : \tau}{E \vdash \lambda.M : \sigma \to \tau} \text{ (cofinie)}$$

L est un ensemble fini de noms qui doivent être «évités».
Typiquement il inclut $Dom(E)$ et $FV(M)$ mais peut être plus grand.

- Pour les récurrences : on sait que la prémisse est vraie pour une infinité de x.
- Pour l'introduction : on peut choisir L suffisamment grand pour exclure tous les x qui pourraient invalider la prémisse.
- Pas besoin de montrer l'équivariance de $E \vdash M : \tau$.

En guise de conclusion

Les résultats du POPLmark challenge

15 solutions, 8 représentations des variables et des lieurs.

Pas de conclusion.

	Alpha Prolog	Coq	Twelf	ATS	Isabelle/HOL	Matita	Abella
de Bruijn		Vouillon, Charguéraud (a)			Berghofer		
HOAS			CMU				Gacek
Weak HOAS		Ciaffaglione and Scagnetto					
Hybrid				Xi			
Locally nameless		Chlipala, Leroy, Charguéraud (b)				Ricciotti	
Named variables		Stump					
Nested abstract syntax		Hirschowitz and Maggesi					
Nominal	Fairbairn				Urban et al.		

Quelques mécanisations de langages et de systèmes de types

Auteurs	Langage	Assistant	α ?
Appel et al	Proof-Carrying Code	Twelf	HOAS
Barras & Werner	Coq en Coq	Coq	dB
Bengtson & Parrow	pi-calcul, psi-calcul	Isabelle	nominal
Crary & Harper	Standard ML	Twelf	HOAS
Charguéraud	mini-ML, Constructions	Coq	LN
Jourdan, Jung, et al	Le langage lambdaRust	Coq	non
Dubois & Menissier	Algorithme W	Coq	non
Nipkow	Algorithme W	Isabelle	non
Nipkow & Urban	Algorithme W	Isabelle	nominal
Pottier & Balabonski	Le langage Mezzo	Coq	dB

Quelques mécanisations de compilateurs

Auteurs	Compilateur	Assistant	α ?
Chlipala	mini-ML + exns + refs $\rightarrow$ asm	Coq	PHOAS
Dargaye	mini-ML $\rightarrow$ Cminor	Coq	dB
Klein & Nipkow	Jinja : Java-light $\rightarrow$ JVM	Isabelle	non
Leroy et al	CompCert : C $\rightarrow$ asm	Coq	non
Myreen et al	CakeML : SML $\rightarrow$ asm	HOL	non

En guise de conclusion

Aucun consensus sur ≪la meilleure≫ approche.

Une tension entre la facilité de représenter un langage objet et la puissance / l'utilisabilité de la logique.

Une curiosité : les approches nominales sont non constructives ?

Une question difficile : l'adéquation (*adequacy*) entre le système qui est formalisé et la compréhension informelle que l'on en a.